A Fault-Tolerant Programming Model for Distributed Interactive Applications

Mogk, Ragnar; Drechsler, Joscha; Salvaneschi, Guido et al.

(2019)

A Fault-Tolerant Programming Model for Distributed Interactive Applications

RAGNAR MOGK, Technische Universität Darmstadt, Germany
JOSCHA DRECHSLER, Technische Universität Darmstadt, Germany
GUIDO SALVANESCHI, Technische Universität Darmstadt, Germany
MIRA MEZINI, Technische Universität Darmstadt, Germany

Ubiquitous connectivity of web, mobile, and IoT computing platforms has fostered a variety of distributed applications with decentralized state. These applications execute across multiple devices with varying reliability and connectivity. Unfortunately, there is no declarative fault-tolerant programming model for distributed interactive applications with an inherently decentralized system model.

We present a novel approach to automating fault tolerance using high-level programming abstractions tailored to the needs of distributed interactive applications. Specifically, we propose a calculus that enables formal reasoning about applications' dataflow within and across individual devices. Our calculus reinterprets the functional reactive programming model to seamlessly integrate its automated state change propagation with automated crash recovery of device-local dataflow and disconnection-tolerant distribution with guaranteed automated eventual consistency semantics based on conflict-free replicated datatypes. As a result, programmers are relieved of handling intricate details of distributing change propagation and coping with distribution failures in the presence of interactivity. We also provides proofs of our claims, an implementation of our calculus, and an empirical evaluation using a common interactive application.

CCS Concepts: • **Theory of computation → Distributed computing models**; • **Software and its engineering → Software fault tolerance**; **Data flow languages**.

Additional Key Words and Phrases: distributed systems, interactive applications, fault tolerance

ACM Reference Format:
Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. 2019. A Fault-Tolerant Programming Model for Distributed Interactive Applications. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 144 (October 2019), 29 pages. https://doi.org/10.1145/3360570

1 INTRODUCTION

Ubiquitous connectivity of web, cloud, mobile, and IoT computing platforms has fostered a variety of distributed interactive applications [Zhang et al. 2014]. Examples of such software systems include collaborative applications (e.g., Trello, Google docs) with real-time aspects (e.g., Uber, Sli.do), remote monitoring and control software (e.g., automated home software, personal health apps), remote processing worksheets (e.g., Jupyter), multi-player gaming, etc.

Such applications often have a decentralized architecture with some components running on cloud platforms and others running on the connected (end-user) devices [Kleppmann et al. 2019]. A decentralized architecture reduces network traffic, improves latency, enhances privacy, and

Authors' addresses: Ragnar Mogk, Technische Universität Darmstadt, Germany; Joscha Drechsler, Technische Universität Darmstadt, Germany; Guido Salvaneschi, Technische Universität Darmstadt, Germany; Mira Mezini, Technische Universität Darmstadt, Germany.

2475-1421/2019/10-ART144
https://doi.org/10.1145/3360570

enables offline usage [Yu et al. 2018], while at the same time supporting collaboration and access to remote resources. In turn, however, a decentralized architecture introduces several failure modes that need to be addressed. First, individual devices may fail (e.g., due to mobiles running out of battery), websites can be reloaded by clients, cloud servers may crash and recover again by restoring persisted state. Second, communication between devices may be temporarily disrupted, causing messages to get lost or be duplicated.

Reasoning about high-level properties of applications running in distributed environments is generally challenging [Gilbert and Lynch 2002]. Significant progress is made in this respect for *data-centric applications* thanks to specialized programming models offering fault-tolerant abstractions. First, there are approaches targeting big data processing on controlled server clusters [Alexandrov et al. 2014; Kreps et al. 2011; Zaharia et al. 2012] that provide high-level programming abstractions with built-in support for recovery of computation's state after crashes. Second, there are approaches targeting computations structured around specialized data structures [Conway et al. 2012; Kuper and Newton 2013; Meiklejohn and Van Roy 2015b], which restrict the programming model in order to tolerate unreliable network conditions [Hellerstein and Alvaro 2019].

Unfortunately, there is no similarly declarative fault-tolerant programming model for distributed *interactive* applications with an inherently decentralized system model. The above approaches do not target applications, where (a) external events and user inputs determine how computations unfold (inversion of control) and (b) individual application components own and operate on data independently. Actor-based approaches [Agha 1986; Akka 2019b; Armstrong 2010] are the state-of-the-art in programming distributed interactive applications. However, they only ensure fault tolerance of individual actors [Bernstein et al. 2014]. Communication failures and crashes affecting multiple actors are left to be handled by the application logic using only low-level message passing.

To recap, for distributed interactive applications, we lack declarative fault-tolerant programming models with easy-to-reason high-level guarantees akin those available for data-centric applications. This is the gap, this work aims to fill. We present a novel approach to automating fault tolerance using high-level programming abstractions tailored to the needs of distributed interactive applications. Specifically, we propose $\mathcal{F}_R$, a formal calculus that enables formal reasoning about applications' dataflow within and across individual devices. $\mathcal{F}_R$ reinterprets the functional reactive programming model [Elliott and Hudak 1997] to seamlessly integrate its automated state change propagation with automated crash recovery of device-local dataflow and disconnection-tolerant distribution with guaranteed automated eventual consistency semantics based on conflict-free replicated datatypes. As a result, $\mathcal{F}_R$ relieves programmers of handling intricate details of distributing change propagation and coping with distribution failures in the presence of interactivity.

In detail, our contributions are as follows:

- We present $\mathcal{F}_R$, a formal model for fault-tolerant interactive applications based on the untyped λ-calculus (Section 3). $\mathcal{F}_R$, which is also informally introduced by examples (Section 2), covers user-defined dynamically reconfigurable dataflow and eventually consistent distribution.
- We prove that the programming model features desirable properties for correctness, determinism, and efficiency in a local setting (Section 4).
- We characterize and prove the fault tolerance properties of our system (Section 4). Devices are restored after a crash and distributed state eventually converges in the presence of both disconnects and crashes.
- We present an implementation of $\mathcal{F}_R$ integrated with the REScala language [Salvaneschi et al. 2014] (Section 5) and demonstrate that our programming model is suitable for typical distributed interactive applications by conducting a case study. We show that our implementation of the case study exhibits similar performance as existing implementations.

```
val name = Source("")
val text = Source("")
val message = (name, text).Map{n => i => append(n, ": ", i)}
val room1 = message.Fold(nilCRDT("topic1")){state => m => consCRDT(state, m)}
val room2 = Source(nilCRDT("topic2"))
val roomList = Source(room1, room2)
val index = Source(0)
val selectedRoom = (roomList, index).Map{l => n => lookup(l, n)}
val roomContent = selectedRoom.Flatten
val contentWidget = roomContent.Map{content => listWidget(content)}
displayUI(name, text, contentWidget)
```

Fig. 1. Complete chat example code for a single device.

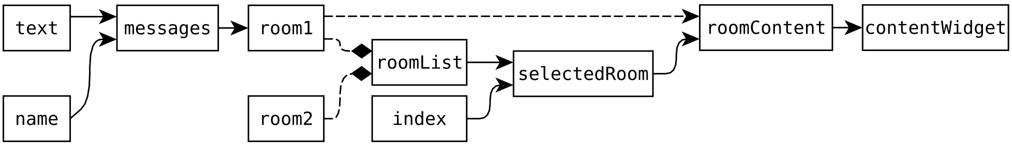

Fig. 2. Dataflow Graph.

2 PROGRAMMING FAULT-TOLERANT INTERACTIVE APPLICATIONS

In this section, we give an informal introduction to the programming model of $\mathcal{F}_R$. We use an example peer-to-peer chat application to illustrate $\mathcal{F}_R$'s abstractions. We use this application as a running example throughout the paper. In general, $\mathcal{F}_R$ supports arbitrary topologies, but, in our example, all devices execute the same application code resulting in a classical peer-to-peer application. Application parts running on different devices communicate via named distributed state, and the language runtime ensures eventual consistency of that state.

2.1 Reactives

In $\mathcal{F}_R$, the basic abstractions are called *reactives* and come in two flavors: source and derived reactives. Sources are discrete time changing values, representing a variety of externally changing values of the device they run on, e.g., user input or sensor data. Derived reactives use *operators* to compute their value from their inputs. In most cases, an operator is a user-defined stateless function, but $\mathcal{F}_R$ also supports a stateful fold and a dynamic flatten reactive. Derived reactives reevaluate their value when their inputs change, forming a *dataflow graph*.

Figure 1 shows the complete code of our running example, which is introduced in this section. Figure 2 shows the dataflow graph of the running example, with each box representing a reactive with their name taken from the variable they are stored in. Arrows point in the direction of dataflow, i.e., the inputs are at the tail of an arrow and the derived reactives are at the tip. Reactives with no incoming arrows are sources. The dashed lines represent dynamic edges, and dashed lines with diamonds represent nesting of reactives in other reactives. In the following, we will incrementally introduce and discuss the application code that builds that graph.

Example 2.1. The code snippet below combines a name and text (source reactives) into a derived chat message. In this case we derive a *map* reactive that appends value of the name and text.

```
12  val text = Source("Hello")
13  val name = Source("Alice")
14  val message = (name, text).Map{n => i => append (n, ": ", i)}
15
16  message.value // ==> "Alice: Hello"
17  text.fire("How are you?")
18  message.value // ==> "Alice: How are you?"
```

The `Source` (Line 12, 13) and `Map` (Line 14) define new reactives. The value of a source is given during creation, and the value of map reactives depends on the current value of its inputs. The current value of reactives is accessible using `.value` (Line 16). The `.fire` (Line 17) is an *action* which changes the value of the source. When an action changes a source, then the derived reactive changes accordingly (Line 18) – a process called *propagation*. Actions are propagated to all transitively derived reactives in the dataflow graph. Thus, `fire` and `value` bridge between the dataflow graph and the surrounding imperative computations expressed in the host language. The host language, used for operators and to create the dataflow graph, is the untyped λ-calculus (with syntactic sugar for the presentation).

Example 2.2. The code extract below models a chat room by collecting all messages into a list using a *fold* reactive. The operator of fold has access to the current value of the fold, thus enabling stateful applications. In this case, whenever `message` changes, then its value `m` is added to the `state` of the fold.

```
19  val room1 = message.Fold(nil){state => m => cons(state, m)}
20
21  room1.value // ==> nil
22  text.fire("1")
23  text.fire("2")
24  room1.value // ==> ("Alice: 1", "Alice: 2")
```

In contrast to map reactives, each time the input of a fold changes, the fold accumulates the value of its input one more time into its current state. Thus, the state of fold must be managed carefully when considering crashes and unreliable networks.

Example 2.3. Next, we introduce `Flatten`, which allows to change edges in the dataflow graph at runtime. We extend our example such that we can use an index to select one of multiple chat rooms to be displayed.

```
25  val roomList = Source((room1, room2)) // list of two rooms
26  val index = Source(0)
27  val selectedRoom = (roomList, index).Map{l => n => lookup(l, n)}
28  selectedRoom.value // ==> 'room1' (a reactive)
29  val roomContent = selectedRoom.Flatten
30  roomContent.value // ==> ("Alice: 1", "Alice: 2"), same as above
```

The `roomList` reactive contains a list of two *nested reactives* and `selectedRoom` selects the one at the position denoted by `index` (Line 27). In Line 29, `roomContent` is defined by flattening `selectedRoom`. This creates a dynamically changing dependency between `roomContent` and either `room1` or `room2`; the dependency changes at runtime as the value of `index` changes.

In Figure 2 the dynamic edge is shown with a dashed arrow and nested reactives are shown as dashed diamonds. In general, flatten reactives enable one layer of indirection when defining dependencies. The flatten reactive (`roomContent`) depends on its input (`selectedRoom`) to select which nested reactive to depend on (e.g., `room1`) and accesses the value of that nested reactive.

2.2 Propagation

A combination of map, fold, and flatten may be used to build stateful and dynamic dataflow graphs for interactive applications. Until now, we have given an intuition that the values in the dataflow graph can be changed, based on external *actions* (e.g., an explicit `fire`). The set of *active* reactives – those directly or indirectly derived from the source of the action – change their values to reflect the action. We say that active reactives *reevaluate* to compute new values in response to an action. The process of *reevaluating* all active reactives is called *propagation* of the action. Conceptually all reevaluations in a single propagation occur at the same time, which defines the semantics of propagation to be synchronous. Synchronous semantics simplifies reasoning about program behavior [Berry and Gonthier 1992; Salvaneschi et al. 2017]. However, in a real implementation, the process of propagating new values to all derived reactives is not instantaneous and external actions or faults may occur in between. Thus, $\mathcal{F}_R$ explicitly models propagation as individual steps and is conceptually very close to actual implementations of propagations algorithms such as SID-UP [Drechsler et al. 2014] and FElm [Czaplicki and Chong 2013]. We formally show that the stepwise propagation behaves equivalently to a synchronous system (Section 4).

2.3 Distribution

$\mathcal{F}_R$ has *distributed reactives* to enable remote communication. Distributed reactives are sources and fold reactives, which replicate their state to other devices. Distributed reactives behave like sources and folds locally. Changes due to remote replication cause actions and start propagation in the same way as local actions.

Example 2.4. We modify our running example to create a distributed chat room, named `"topic1"`. Messages are accumulated into `distributedRoom` as before but using a special list data type for synchronization. The state of this fold will be synchronized with all other devices that also have a distributed fold with the name `"topic1"`.

```
31 val distributedRoom = message.Fold(nilCRDT("topic1")){
32   state => m => consCRDT(state, m) }
```

$\mathcal{F}_R$ uses eventually consistent replicated data types (state-based CRDTs [Shapiro et al. 2011a]) for distributed reactives. CRDTs ensure that the state of distributed reactives can always be synchronized, even if connections to other devices are unreliable or changes are made while offline. CRDTs are data types, whose state domain together with a merge operation defines a lattice, and operations on the state must always produce a state that is larger according to that lattice (c.f. [Shapiro et al. 2011b] for a formal discussion). Existing research shows that many common data types can be expressed as CRDTs [Shapiro et al. 2011a]. Their wide-spread usage confirms the practicality of CRDTs, to replicate data over unreliable connections [Bernstein et al. 2017; Shapiro et al. 2018].

$$\begin{aligned}
lookup(r, \Sigma, v) &= \begin{cases} \Sigma(r) & r \in \Sigma \\ v & r \notin \Sigma \end{cases} \\
ready(P, \mu) &= \{r \in \mu | r \notin P \wedge inputs(r, \mu) \subseteq P \cup \{r\}\} \\
outdated(A, \mu) &= \{r \in \mu | r_i \in A \wedge r_i \in inputs(r, \mu)\} \\
reevexp(r, \mu) &= \mu(r).op\ inputs(r, \mu).val \\
update(\mu, r, v) &= (\mu, r \mapsto (\mu(r).in, v, \mu(r).op)) \\
inputs(r, \mu) &= \begin{cases} \{r_0, \mu(r_0).val\} & \mu(r).in = dynamic(r_o) \\ \mu(r).in & \text{otherwise} \end{cases} \\
location(\mu) &= \text{fresh location in } \mu\text{, consistent between restarts}
\end{aligned}$$

Fig. 3. Auxiliary functions for the operational semantics.

Conceptually, any local fold operation is expressible as a distributed reactive by representing all changes to fold as an ordered list and distributing that list (i.e., as in Example 2.4), and then folding that list independently on each device. However, because finding efficient CRDT implementations remains and active field of research [Almeida et al. 2018; Enes et al. 2019], $\mathcal{F}_R$ allows arbitrary (state-based) CRDTs for distributed reactives, which have different trade-offs concerning supported operations and synchronization performance. The implementation of $\mathcal{F}_R$ comes with a library of CRDTs for the developer to select from. In Section 3, we show that the composition of distributed reactives is still eventually consistent, a result that is not always true for the composition of plain CRDTs [Meiklejohn and Van Roy 2015b; Wang et al. 2019].

2.4 Restoration

Each device in $\mathcal{F}_R$ has its own application code together with its own storage for state. When a crash occurs, the runtime representation of the dataflow graph is lost including all values of reactives. The dataflow graph itself can be reconstructed from the application code, but values of reactives include user data that cannot be reconstructed. $\mathcal{F}_R$ stores snapshots of the values of reactives on permanent storage of the device. Using the application code and the snapshot, the dataflow graph and all values are fully restored after a crash.

To minimize lost state, $\mathcal{F}_R$ updates the snapshot after every processed action, thus losing at most the latest interaction with the user (i.e., a single press of a button). To ensure efficiency of interactions, $\mathcal{F}_R$ minimizes the work for each snapshot. First, snapshots are stored incrementally, only including values changed by the action. Second, there is no need to store the values of map and flatten reactives, as they can be recomputed. Thus, only values which cannot be recomputed – `sources` and `folds` – are included in the snapshot. All other values are lazily recomputed during restoration, trading efficient frequent snapshots for more expensive but rare restorations. Section 4 elaborates on further details and proves the correctness of restoration from minimal snapshots.

Example 2.5. For illustrating the restoration semantics, assume the user had sent some messages while offline (i.e., submitting text from the UI), which are now stored in `room1.value = ("msg1", "msg2", ...)`. Every time the user generates a new message, the runtime updates the snapshot of `room1` to the new value. After a crash, instead of initializing `room1` with an empty list CRDT as defined by the application code, $\mathcal{F}_R$ loads the value from the snapshot, restoring `room1` to contain all messages (including those sent while offline). Once the runtime reconnects, the restored messages are sent to the network, thus no state is lost, and consistency is ensured by the CRDT semantics.

$$
\begin{array}{rll}
C & ::= (M; S; D|\ldots|D) & \textit{Communication} \\
D & ::= \Sigma L & \textit{Devices} \\
L & ::= \langle\mu\rangle\, t \mid\ \triangleright(U\ \langle\mu\rangle\, t) & \textit{Processes} \\
U & ::= t \cdot r \cdot (A; P) \mid (A; P) & \textit{Updates} \\
v & ::= x \Rightarrow t \mid r \mid \text{unit} & \textit{Values} \\
t & ::= x \mid x \Rightarrow t \mid t\ t \mid \text{unit} \mid & \lambda\ \textit{Terms} \\
 & \quad r \mid \text{Source}(t) \mid (t, t).\,\text{Map}\{t\} \mid t\,.\,\text{Fold}(t)\{t\} \mid t.\,\text{Flatten} \mid & \textit{Reactives} \\
 & \quad t.\text{value} \mid t.\text{fire}(t) & \textit{Actions}
\end{array}
$$

$$
\begin{array}{ll}
M : r \mapsto ((v \times v) \to v) & \textit{Merge functions} \\
S : r \times v & \textit{Messages} \\
\Sigma : r \mapsto v & \textit{Snapshots} \\
\mu : r \mapsto (in \in \mathcal{P}(r), val \in v, op \in v) & \textit{Local stores}
\end{array}
$$

Fig. 4. Syntax of $\mathcal{F}_R$.

2.5 Summary

$\mathcal{F}_R$ seamlessly integrates eventually consistent data types and crash tolerance into the synchronous semantics of dataflow programming models. The integration is achieved by basing all abstractions on a single notion of reactives with state. Moreover, through the integration crashes and disconnects are automatically tolerated in a programming model that is easier to reason about for programmers. The following sections elaborate on how $\mathcal{F}_R$ achieves this using an operational semantics to give a precise definition.

3 SYNTAX AND SEMANTICS

Syntax. Figure 4 shows the syntax of $\mathcal{F}_R$. The model for communication of interactive application $(M; S; D_1|\ldots|D_n)$ consists of a mapping M assigning distributed reactives a merge function $r \mapsto merge_r$ to model CRDTs, a set of in-flight messages S containing pairs of reactives and values (r, v), and a set of devices $D_1|\ldots|D_n$ executing concurrently. Devices D have volatile state in the form of processes L, and durable state in the form of snapshots Σ. The current process is lost when the device crashes, but the snapshot is persisted. Each D_i starts with an initial process L_i and an empty snapshot.

The evaluation of processes L models either (a) the execution of application code $\langle\mu\rangle\, t$ with term t and store μ or (b) the propagation of actions $\triangleright(U\ \langle\mu\rangle\, t)$. The evaluation of processes is based on λ-calculus. Values v include functions $x \Rightarrow t$, the unit value, and reactives r. Terms t include function definitions and application, creating new reactives, and reading and firing reactives. In $\mathcal{F}_R$ all terms creating new reactives are capitalized. Reactives that are used as inputs or for activation are left of a dot, and user-defined operators for reevaluation are in braces. We use a standard, left to right, call by value, evaluation context E [Felleisen and Hieb 1992]. The store μ maps reactives r to a 3-tuple with named values containing the set of input reactives $(in \in \mathcal{P}(r))$, the current value $(val \in v)$, and the operator used for reevaluation $(op \in v)$. We write $\mu(r).val$ to access the value val of reactive r in store μ. The current update U is used by the runtime for bookkeeping during propagation of actions. U contains the sets of active A and processed P reactives, and optionally a reactive operator t for reactive r.

$\boxed{\rightarrow_D \subset D \times D}$

$$\frac{\mu(r).in = \emptyset}{\Sigma\,\langle\mu\rangle\, r.\text{fire}(v) \rightarrow_D \Sigma \triangleright ((\{r\};\{r\})\,\langle update(\mu, r, v)\rangle\, r)}\ (\textsc{fire})$$

$$\frac{P = dom(\mu) \qquad \Sigma' = \Sigma, \{\mu(r).val \mid r \in A \land (r \in \mu(r).in \lor \mu(r).in = \emptyset)\}}{\Sigma \triangleright ((A; P)\,\langle\mu\rangle\, t_l) \rightarrow_D \Sigma'\,\langle\mu\rangle\, t_l}\ (\textsc{commit})$$

$$\frac{v = \mu(r).val}{\Sigma\,\langle\mu\rangle\, r.\text{value} \rightarrow_D \Sigma\,\langle\mu\rangle\, v}\ (\textsc{access}) \qquad \frac{t \rightarrow_\lambda t'}{\Sigma\,\langle\mu\rangle\, t \rightarrow_D \Sigma\,\langle\mu\rangle\, t'}\ (\textsc{outer})$$

$$\frac{\Sigma\,\langle\mu\rangle\, t \rightarrow_D \Sigma\,\langle\mu'\rangle\, t'}{\Sigma\,\langle\mu\rangle\, E[t] \rightarrow_D \Sigma\,\langle\mu'\rangle\, E[t']}\ (\textsc{context})$$

Fig. 5. Operational semantics for process behavior of devices.

Compared to the syntax presented in this section, the syntax used in Section 2 has the following variations, adopted for readability. (a) A flexible number of input reactives for `Map`, (b) `val x = t` for binding `x` to `t` in the rest of the block, (c) use of literals like numbers (`10`), Strings (`"string"`), and lists (`t, t, t`), (d) predefined functions (`append`, `cons`), (e) parenthesis to clarify associations, (f) line comments (`//`), and (g) sequencing multiple terms separated by newlines.

Semantics overview. We define a small step operational semantics for $\mathcal{F}_R$. Figure 3 contains auxiliary functions used throughout the semantics. Small step semantics allows modeling of message receives and crashes to nondeterministically occur between any two steps of a device. We first look at the stepping rules for individual devices (Section 3.1). Evaluation of a single device depends on the current process, and is separated into two differently labeled stepping relations, $\rightarrow_D$ and $\rightarrow_p$, for presentation purposes. When evaluating application code ($\rightarrow_D \subset D \times D$) the dataflow graph is created (Section 3.2) and actions are received. An action starts a propagation ($\rightarrow_p \subset D \times D$) (Section 3.3). Propagation processes all reactives before normal execution of the application continues. A crash of the device during a propagation causes the triggering action to be lost. We extend the system to many communicating devices in Section 3.4. Communication has its own stepping relation ($\rightarrow_C \subset C \times C$), which nondeterministically chooses to further evaluate one of the devices, to send messages, or to receive messages. We make the usual fairness assumptions, that eventually all devices get the chance to evaluate and communicate.

3.1 Devices

A device $D = \Sigma L$ consists of a currently executing process L, and a snapshot Σ. A snapshot Σ is a mapping $(r \mapsto v)$ from reactives to their value. Figure 5 shows the device evaluation relation except creating reactives.

Processes have two syntactic forms: (1) $\langle\mu\rangle\, t$ executes the application code t with store μ to create and modify the dataflow graph, and (2) $\triangleright(U\,\langle\mu\rangle\, t)$ executes the runtime propagation of changes caused by actions, where $\langle\mu\rangle\, t$ is the application level code to continue with after the update, and U contains the state of the runtime update propagation. The FIRE and COMMIT rules switch between the two forms of processes.

The FIRE rule evaluates the term $r.\text{fire}(v)$ to create a new action on source r with value v and activates r, i.e., marks it as having changed. The precondition ensures that r has no inputs, i.e., it is a source. The device switches to propagation by adding the runtime state $(\{r\}; \{r\})$, which is

$$\boxed{\longrightarrow_D \subset D \times D}$$

$$\frac{r = location(\mu) \qquad v_s = lookup(r, \Sigma, v)}{\Sigma\ \langle \mu \rangle\ \text{Source}(v) \longrightarrow_D (\Sigma, r \mapsto v_s)\ \langle \mu, r \mapsto (\emptyset, v_s, \text{unit}) \rangle\ r}\ (\textsc{source})$$

$$\frac{r = location(\mu) \qquad \mu' = \mu, r \mapsto (\{r_1, r_2\}, \text{unit}, v_{op}) \qquad t = reevexp(\mu', r)}{\Sigma\ \langle \mu \rangle\ (r_1, r_2).\,\text{Map}\{v_{op}\} \longrightarrow_D \Sigma\ \triangleright (t \cdot r \cdot (\{r\}; \{r\})\ \langle \mu' \rangle\ r)}\ (\textsc{map})$$

$$\frac{r = location(\mu) \qquad v_s = lookup(r, \Sigma, v)}{\Sigma\ \langle \mu \rangle\ r_1\,.\,\text{Fold}(v)\{v_{op}\} \longrightarrow_D (\Sigma, r \mapsto v_s)\ \big\langle \mu, r \mapsto (\{r, r_1\}, v_s, v_{op}) \big\rangle\ r}\ (\textsc{fold})$$

$$\frac{r = location(\mu) \qquad \mu' = (\mu, r \mapsto (\text{dynamic}(r_0), \mu(\mu(r_0).val).val, x_r \Rightarrow x_v \Rightarrow x_v))}{\Sigma\ \langle \mu \rangle\ r_0.\,\text{Flatten} \longrightarrow_D \Sigma\ \langle \mu' \rangle\ r}\ (\textsc{flatten})$$

Fig. 6. Operational semantics for creating reactives.

read as r is *active* (it has changed its values) and r was *processed* (it will not change anymore in the current propagation). Section 3.3 explains how these sets are used for propagation. The current term of the process r.fire(v) evaluates to just r, indicating that firing a source produces no useful new value. Finally, the value of the state $\mu(r)$ is changed to v – we write $update(\mu, r, v)$ to represent the updated store (c.f. Figure 3).

The COMMIT rule ends the propagation and updates the snapshot to include all changes. When a propagation ends, the set of processed reactives P contains all reactives in the domain of μ. The COMMIT rule steps from a propagation that processed all reactives back to normal application execution by keeping the term and store $\langle \mu \rangle\ t_l$ from the propagation and producing a new snapshot Σ' which reflects the changes to folds and sources A that were active in the propagation. We write $\Sigma' = (\Sigma, r \mapsto v)$ to say that Σ' contains the same value assignments as Σ, except for r which is updated to v. The premise of COMMIT asserts that Σ' is updated to contain the current values of all active sources and folds $r \in A$. Sources and folds are computed by inspection of the inputs $\mu(r).in$, sources have no inputs, and folds have themselves as an input.

The ACCESS rule evaluates the term to the current value of the reactive, without modifying the store nor the snapshot.

The OUTER rule embeds the stepping relation of the λ-calculus outside of reactive operators. The rules of the *lambda*-calculus are not shown but are standard call-by-value rules using substitution.

The CONTEXT rule evaluates nested device terms.

3.2 Creating and Restoring the Dataflow Graph

The reduction rules in Figure 6 are concerned with creating the dataflow graph. The SOURCE, MAP, FOLD, and FLATTEN rules each create a fresh identifier $r = location(\mu)$, and add the inputs, value, and operators $r \mapsto (in, val, op)$ to the store μ. Restoration from a snapshot happens during the creation of sources and folds, and changes their initial values depending on the snapshot.

The SOURCE rule creates a reactive r with initial value v_s and neither operator nor inputs. If the snapshot contains r, then the value is restored from the snapshot, i.e., $v_s = \Sigma(r)$. If the snapshot does not contain r, then the given value is used, i.e., $v_s = v$.

The MAP rule derives a new reactive from two inputs r_1, r_2, with operator v_{op}. Map reactives must evaluate their operator on their inputs to compute their initial value. The initial evaluation uses the same mechanism as the propagation of actions. A new propagation with reevaluation

$$
\begin{aligned}
\mu =& (\\
r_{name} \mapsto& (\emptyset, \texttt{"some name"}, \text{unit}), \\
r_{text} \mapsto& (\emptyset, \texttt{"some message"}, \text{unit}), \\
r_{message} \mapsto& (\{r_{name}, r_{text}\}, \texttt{"some name: some message"}, \texttt{n => i => append (n, ": ", i)}), \\
r_{topic1} \mapsto& (\{r_{topic1}, r_{message}\}, \texttt{nilCRDT("topic1")}, \texttt{state => m => consCRDT (m, state)}), \\
r_{topic2} \mapsto& (\emptyset, \texttt{nilCRDT("topic2")}, \text{unit}), \\
r_{roomList} \mapsto& (\emptyset, \texttt{(}r_{room1}\texttt{, } r_{room2}\texttt{)}, \text{unit}), \\
r_{index} \mapsto& (\emptyset, \texttt{0}, \text{unit}), \\
r_{selectedRoom} \mapsto& (\{r_{roomList}, r_{index}\}, r_{topic1}, \texttt{l => n => lookup (l, n)}), \\
r_{roomContent} \mapsto& (\{\text{dynamic}(r_{selectedRoom})\}, \texttt{nilCRDT("topic1")}, \texttt{unit}) \\
&)
\end{aligned}
$$

```
10  val contentWidget = r_roomContent.Map {content => listWidget (content)}
11  displayUI(r_name, r_text, contentWidget)
```

Fig. 7. Example of the store after evaluating the chat example up to line 10.

$t \cdot r$ starts. The term $t = reevexp(\mu', r)$ computes the application of the operator v_{op} to the inputs r_1, r_2. The propagation initiated by the MAP rule will always only change r, because r does not have any other derived value yet. Map reactives are neither stored nor restored from the snapshots. The operator v_{op} is applied to the restored values of r_1, r_2 to compute the restored value, because operators in $\mathcal{F}_R$ (i.e., λ-calculus terms) are deterministic.

The FOLD rule creates a reactive r with operator v_{op}. The initial value v_s is restored in the same way as in the SOURCE rule. To model access of its state during reevaluation r has itself as an input in addition to r_0, thus passing its own value to its operator.

The FLATTEN rule creates reactives which derives its value from a single input reactive r_0. The initial value $\mu(\mu(r_0).val).val$ of flatten precisely describes the indirection flatten creates. However, flatten reactives have a specially labeled input dynamic(r_0), instead of just r_0. Propagation requires special support for handling graphs with dynamic(r_0) dependencies described in Section 3.3. The $reevexp(\mu', r)$ of flatten applies two values to the operator: (1) the value of r_0 (the outer reactive) named x_r, and (2) the value of $\mu(r_0).val$ (the inner reactive) named x_v. Thus, the operator of flatten returns the value of the inner reactive x_v. Like map reactives, flatten does not directly restore its values, but all inputs of flatten are assumed to be restored before the flatten is created.

Storing nested reactives. Storing a reactive inside a source or fold reactive requires that reactive to be stored in the snapshot for restoration. For storing a snapshot Σ, this means that if the value v_0 of reactive r_0 is included in the snapshot, then also any other reactive r referenced from that v_0 must have its value stored in Σ. This process is recursive. During restoration of r_0, the values of any inner r are also restored, if r has not yet been restored otherwise. Neither the operator nor the inputs of r are restored however, and the reactive is essentially constant. In the case that r is recreated later, the restored value will be overwritten, and the inputs connected. We will show in Section 5 how the restoration of nested reactives is achieved in a practical implementation.

Example 3.1. We now have all constructs required to build a dataflow graph of an $\mathcal{F}_R$ application. Figure 7 shows store μ (which encodes the dataflow graph) and the remaining application term

$$\boxed{\rightarrow_p \subset D \times D}$$

$$\frac{r \in \mathit{ready}(P, \mu) \qquad r \notin \mathit{outdated}(A, \mu)}{\Sigma \triangleright ((A; P)\,\langle\mu\rangle\, t_l) \xrightarrow{r}_p \Sigma \triangleright ((A; P, r)\,\langle\mu\rangle\, t_l)}\ (\textsc{skip})$$

$$\frac{r \in \mathit{ready}(P, \mu) \qquad r \in \mathit{outdated}(A, \mu) \qquad t = \mathit{reevexp}(r, \mu)}{\Sigma \triangleright ((A; P)\,\langle\mu\rangle\, t_l) \xrightarrow{r}_p \Sigma \triangleright (t \cdot r \cdot (A, r; P, r)\,\langle\mu\rangle\, t_l)}\ (\textsc{reevaluate})$$

$$\frac{t \rightarrow_\lambda t'}{\Sigma \triangleright (t \cdot r \cdot (A; P)\,\langle\mu\rangle\, t_l) \xrightarrow{r}_p \Sigma \triangleright (t' \cdot r \cdot (A; P)\,\langle\mu\rangle\, t_l)}\ (\textsc{inner})$$

$$\frac{}{\Sigma \triangleright (v \cdot r \cdot (A; P)\,\langle\mu\rangle\, t_l) \xrightarrow{r}_p \Sigma \triangleright ((A; P)\,\langle \mathit{update}(\mu, r, v)\rangle\, t_l)}\ (\textsc{write})$$

Fig. 8. Operational semantics for propagation and reevaluation.

after evaluating up to Line 10 of Figure 1. The application was structured in a way, that each created reactive was assigned a variable, and we have reused these names to label the location of each reactive for better understanding, except for the distributed reactives which use their given names as location names.

3.3 Runtime Processing of Actions

The rules for propagation and reevaluation $\xrightarrow{r}_p$ of reactive r are shown in Figure 8. We write just $\rightarrow_p$ if the specific reactive is not important. Whenever an action changes the value of a reactive the runtime starts *propagation* of that change and all transitively derived reactives must *reevaluate*, i.e., compute their new value based on the inputs and on the operator. Syntactically, processes doing propagations are written $\triangleright(U\ \langle\mu\rangle\ t)$. With U containing the state of the propagation including the set of active A and processed P reactives. $\rightarrow_p$ evaluates devices with such processes. From the application developers' point of view, all reevaluations happen at the same time and use the most up-to-date value of their inputs. $\mathcal{F}_R$ models propagation as a stepwise process to reason about failure cases but guarantees synchronous semantics. At the beginning of a propagation, an action has changed the value of a reactive r, which is active $r \in A$ and processed $r \in P$. During the propagation, ready reactives $r' \in \mathit{ready}(\mu, P)$ are either reevaluate or skipped, until all reactives are processed.

Reevaluation. A reevaluation is the process of computing the current value of a reactive r using the operator op and the inputs of r by evaluating $t = \mathit{reevexp}(r, \mu)$ (Figure 3) which applies op to the values of the inputs of r. We write $\mathit{inputs}(r, \mu).\mathit{val}$ to access the value of all inputs of r and apply the operator in the correct order to the values. In case r is a flatten reactive the $\mathit{inputs}(r, \mu)$ function accesses the nested reactive, which is then treated as a normal input.

Reevaluation starts when an expression t and a reactive r are added to the propagation configuration U by the reevaluate rule. The inner rule evaluates the expression t_2 according to $\rightarrow_\lambda$ until it is a value. The resulting value v is written by the write rule as the new value of the reactive r in the store μ.

Propagation. The initial activation caused by the fire rule, writes a new value to the store μ and marks r as active and processed. A reactive r is ready $r \in \mathit{ready}(P, \mu)$ if it has not been processed and all inputs of r are processed. Fold reactives have themselves as inputs and are ready if all other

$$\boxed{\rightarrow_C \subset I \times I}$$

$$\frac{D_i \rightarrow_D D_i'}{(M; S; D_1|\ldots|D_i|\ldots|D_n) \rightarrow_C (M; S; D_1|\ldots|D_i'|\ldots|D_n)}\ (\textsc{device})$$

$$\frac{r \in dom(M) \qquad r \in dom(\mu)}{(M; S; D_1|\ldots|\Sigma\,\langle\mu\rangle\, t_l|\ldots|D_n) \rightarrow_C (M; S, (r, \mu(r).val); D_1|\ldots|\Sigma\,\langle\mu\rangle\, t_l|\ldots|D_n)}\ (\textsc{send})$$

$$\frac{}{(M; S, (r, v_r); D_1|\ldots|D_n) \rightarrow_C (M; S; D_1|\ldots|D_n)}\ (\textsc{drop})$$

$$\frac{\begin{array}{c} merge = M(r) \qquad r \in dom(\mu) \qquad v = merge(\mu(r).val, v_r) \\ D_i = \Sigma\,\langle\mu\rangle\, t_l \qquad D_i' = \Sigma \triangleright ((\{r\}; \{r\})\,\langle\mu(r).val \leftarrow v\rangle\, t_l) \end{array}}{(M; S, (r, v_r); D_1|\ldots|D_i|\ldots|D_n) \rightarrow_C (M; S; D_1|\ldots|D_i'|\ldots|D_n)}\ (\textsc{receive})$$

Fig. 9. Operational semantics for remote updates.

inputs are processed. Flatten reactives are ready when the outer and the inner inputs are processed. Additionally, a reactive r is outdated $r \in outdated(A, \mu)$, if any of its inputs are active, i.e., the input has been modified. Depending on whether a ready reactive is outdated the SKIP or REEVALUATE rules are applied. The SKIP rule just marks a reactive as processed, if it is ready and not outdated. The REEVALUATE rule additionally causes a reevaluation of the reactive and marks the reactive as active. In both cases, the reactive is marked as processed. When all reactives in the store have been processed the COMMIT rule ends the propagation. Due to both sets, active reactives A and processed reactives P, only growing during a propagation, the process is guaranteed to terminate.

3.4 Communication between Multiple Devices

Communication is based sending state between devices without requiring ordering or reliability of messages, thus the guarantees of $\mathcal{F}_R$ apply to most existing systems. The stepping rules for communicating devices $(M; S; D_1|\ldots|D_n)$ are shown in Figure 9. The merge functions in M define the global behavior of the distributed reactives and they are fixed before any device starts – no central coordination is required. The messages in S model an unreliable communication channel, with reordered, duplicated, and lost messages.

The DEVICE rule models any of the concurrent devices D_i taking a normal step in the device evaluation relation $\rightarrow_D$ (c.f., Section 3.1). This model allows for devices to execute at different speeds or pause execution for some time before resuming.

In the SEND rule a device $D_i = \langle\mu\rangle\, t_l$, which has a reactive r with value $v_r = \mu(r).val$ and an associated merge function $M(r)$, sends message (r, v_r). We do not model specific targets for messages but assume that messages are eventually sent to every other interested device at least once. The DROP rule removes a message from the set of messages.

The RECEIVE rule models the device $D_i = \langle\mu\rangle\, t_l$, receiving a remote value v_r for reactive r. $(S, (r, v))$ is the disjoint union, i.e., $\{(r, v)\}$ is separated from S thus consuming the received message. The received remote value v_r is merged with the local value $\mu(r).val$ resulting in $v = merge(\mu(r).val, v_r)$. D_o then starts a propagation using $r \mapsto v$ as the update. Sending and receiving may happen at any time when there is no ongoing propagation.

To ensure eventual consistency between devices, the merge function must form a semilattice (i.e., it is associative, commutative, and idempotent). Additionally, we only allow fold reactives and sources (i.e., reactives which already have state) to be updated remotely. However, eventual consistency does require that all devices eventually do receive new messages for all reactives.

Sending nested reactives. Distributed reactives r_0 may contain other nested reactives r as values. When a device D adds r to the value of r_0 for the first time, there are two cases to consider. First, if r is already a distributed reactive each device uses its local value of r inside of r_0. The two reactives r and r_0 are synchronized independently. Second, if r is not a distributed reactive, only the local device d can cause r to change. In this case, d promotes r to a distributed reactive by providing an initial value and a merge function. The merge function for r selects the latest value according to a logical timestamp. This latest-writer-wins scheme is race condition free, because the original device is the only writer of r. The current value of r is sent along the value of r_0 when synchronizing, to allow remote devices to initialize a replica of r. Once initialized, r synchronizes independently of r_0. In the next section, we assume that this transformation has already been applied for each nested reactive.

4 THEORY OF DEVICES, DISTRIBUTION, AND RESTORATION

In this section, we first present the results about the evaluation of individual devices. We assume that user-defined operators always terminate, i.e., they only contain terminating λ-calculus terms. We will show that update propagation on a single device is glitch-free, complete, deterministic, and isolated. These local guarantees form the foundation for fault tolerance for distributed state and restoration for local state.

Definition 4.1 (Syntax). We write $\longrightarrow = \longrightarrow_D \cup \longrightarrow_p$, and $\longrightarrow^*$ for the transitive closure. We write $\mu \in D$ to say that μ store is the store of D, and similar with other syntax. We write $D \in D_0 \longrightarrow \ldots \longrightarrow D_n$ and $D_i \longrightarrow D_j \in D_0 \longrightarrow \ldots \longrightarrow D_n$ to say that D and $D_i \longrightarrow D_j$, respectively, are contained in the sequence $D_0 \longrightarrow \ldots \longrightarrow D_n$.

4.1 Traces

We use *traces* to reason about device evaluation represented as sequences of steps, e.g., a single propagation or the initialization of the application. We generally assume that traces are finite, i.e., that the evaluation of devices terminates when no external inputs are received.

Definition 4.2 (Trace). A trace of a device D_0, written $\mathit{trace}(D_0)$ is a sequence $D_0 \longrightarrow^* D_n$, where D_n cannot be further reduced.

Definition 4.3 (Propagation). Given D_0, a propagation $p = \mathit{ptrace}(D_i, D_j)$ is any maximally long subsequence $D_i \longrightarrow_p^* D_j$ of $\mathit{trace}(D_0)$, i.e., there is no longer sequence $D_i' \longrightarrow_p^* D_j'$ that contains $D_i \longrightarrow_p^* D_j$.

Definition 4.4 (Reevaluation). We call any $D \overset{r}{\longrightarrow}_p D'$ which was produced by the REEVALUATE rule, a reevaluation of r.

4.2 At-Most-Once Reevaluation of Reactives

In a realistic implementation, multiple reevaluations of a single reactive are observable by user code using side effects. Even in $\mathcal{F}_R$, multiple reevaluations would produce incorrect values for folds, i.e., the inputs are aggregated multiple times. We show that propagation ensures that reactives are reevaluated at-most-once.

Lemma 4.5 (At-most-once evaluation). *Each propagation $ptrace(D_0, D_n)$ contains at most one reevaluation $D \xrightarrow{r}_p D'$ of each reactive r.*

Proof. By the premise of reevaluate it must be $r \in ready(P, \mu)$ which requires $r \notin P$. However, the reevaluate rule adds r to P, and no step during a propagation removes reactives from P. Thus, there can be at most one reevaluation of each reactive r. □

4.3 Glitch-Free Propagation

When actions are incorrectly propagated, inconsistencies – called glitches – can arise where a mix of values before and after the action is observed. A system is called glitch-free if no glitches are observable by user code. In the case of $\mathcal{F}_R$, only user-defined operators observe multiple values of a single propagation (c.f., isolation in Lemma 4.13), thus potentially observe a glitch. An operator observes a glitch (mixed values) if one of its inputs is written before the other input after the reevaluation of the operator in a single propagation.

Lemma 4.6 (Glitch Freedom). *For any propagation $p = ptrace(D_0, D_n)$, and reevaluation $D_{j-1} \xrightarrow{r}_p D_j$ in p and for all inputs $r' \in inputs(r, \mu) \setminus \{r\}$ of r, there is no write $D_{k-1} \xrightarrow{r'}_p D_k$ with $k > j$ in p.*

Proof. By contradiction. Assume there is a write on r' satisfying the conditions above. Because r is reevaluated it must be $r \in ready(P, \mu)$ for some μ and P, thus $r' \in P$ at the time of the reevaluation. Due to at-most-once reevaluation (Lemma 4.5), for $r' \in P$ to be true, the sole reevaluation of r' at $D_{i-1} \xrightarrow{r}_p D_i$ it must be $j > i$, i.e., r' is reevaluated before r. However, $k > j > i$ means that the reevaluation of r is between the reevaluation of r and the write of r, which by inspection of the rules is impossible.

□

4.4 Complete Propagation

At-most-once evaluation (Section 4.2), and glitch freedom (Section 4.3) are trivially fulfilled if actions are not propagated at all and hence, we require an additional liveness property. We show that after a propagation, all derived reactives reflect the changes to their inputs, given their operators. For folds applying their operator multiple times aggregates new values even without changed inputs, thus it is also incorrect to just reevaluate all reactives. We show that $\mathcal{F}_R$ reevaluates reactives that are reachable in the dataflow graph from the action (Lemma 4.10). To this end, we first show that there is always a *ready* reactive until all reactives become processed, and exactly the reachable reactives become active (Lemma 4.8).

Lemma 4.7. *For any step during a propagation, either all reactives are processed $P = dom(\mu)$, or there is at least one $r \in ready(P, \mu)$.*

Proof. By construction. Pick any unprocessed $r \in dom(\mu) \setminus P$, if r is ready, we found a candidate. Otherwise, there must be an unprocessed input $r_i \in inputs(r, \mu)$ from which we continue our search. This search must terminate, because the graph is acyclic. □

Lemma 4.8. *At the end of any propagation started by an action on reactive r the set of active reactives A is the set of transitively derived reactives of r.*

Proof. Inspection of the rules show that r is added to A at the start of a propagation (fire, remote). Reactives are *outdated* only if they are derived from reactives in A. Exactly the *outdated* and *ready* reactives are processed by the reevaluate rule which adds them to A. All other *ready* reactives are processed by the skip rule, which does not add them to A. Because of Lemma 4.7 there is always a ready reactive until all reactives are added to P and reachable ones are also added to A. □

Definition 4.9 (Complete Reactives). Given a propagation $p = ptrace(D, D')$ with respective stores $\mu \in D$ and $\mu' \in D'$, and new synthetic stores $\mu'_r = update(\mu', r, \mu(r).val)$. We say a reactive r is $complete(r)$ in p if and only if reevaluating r in μ_r produces $reevexp(r, \mu_r) \rightarrow^*_\lambda \mu'(r)$.

Lemma 4.10 (Complete Propagation). *For any propagation p starting with an action setting r to v and ending in store μ', we call p complete if*

- *the action changes the correct reactive $\mu'(r).val = v$,*
- *all active derived reactives $\{r' \in A\}$ are complete,*
- *all non-active folds and sources keep their values.*

Proof. First, the rules causing an action – fire and receive – set the correct state and mark the reactive as processed, and because of at-most-once reevaluation (Lemma 4.5) we know those will not be changed again. Second, all active reactives are reevaluated (Lemma 4.8). Due to glitch freedom (Lemma 4.6) each reevaluated reactive is complete, and because of at-most-once evaluation (Lemma 4.5) they do not change afterwards. And last, the skip rule causes all non-active reactives to become processed without changing their value (Lemma 4.8). □

4.5 Determinism

While the order of reevaluations during propagation is not deterministic propagation is confluent, i.e., always produces the same result. Thus, we say execution of devices is deterministic.

Lemma 4.11 (Confluence). *For any two propagations starting at the same configuration $p_1 = ptrace(D, D_1)$ and $p_2 = ptrace(D, D_2)$, the final states $\mu_1 \in D_1$ and $\mu_2 \in D_2$ are equal $\mu_1 = \mu_2$.*

Proof. No reactives are created during a propagation, thus $dom(\mu) = dom(\mu_1) = dom(\mu_2)$. Due to Lemma 4.8 the set of active reactives $A_1 \in D_1$ and $A_2 \in D_2$ are the same $A_1 = A_2$, and by complete propagation (Lemma 4.10) the evaluate to the same values. □

Lemma 4.12 (Determinism). *For any device D all execution traces $trace(D)$ contain the same sequence of $\rightarrow_D$ steps, i.e., they are equal after removing all $\rightarrow_p$ steps.*

Proof. Follows from the determinism of the λ-calculus, which we extend only with deterministic $\rightarrow_D$-rules, and the confluent propagation (Lemma 4.11). □

4.6 Isolated Propagation

Isolation captures the final piece of the synchronous nature of $\mathcal{F}_R$ by stating that propagations do not interfere with each other. $\mathcal{F}_R$ itself executes only one propagation at a time, thus propagations are trivially isolated from each other. In a distributed $\mathcal{F}_R$ application, propagations are executed concurrently. However, because there is no shared state, propagations on different devices are naturally isolated from each other.

Lemma 4.13 (Isolation). *The resulting configuration D' of a propagation $p = ptrace(D, D')$ is independently of any concurrent action.*

Proof. Propagation is confluent (Lemma 4.11) and there are no concurrent actions by inspection of the rules. Neither the fire nor remote rule may trigger a concurrent action when a propagation is in progress. □

4.7 Optimal Parallelization of Propagation

$\mathcal{F}_R$ is designed for efficient implementations by allowing for high level optimizations due to the use of managed propagation. Reevaluation of multiple reactives which are *ready* are parallelizable, and we show that the algorithm used in $\mathcal{F}_R$ is optimal with regards to the number of reactives that are *ready* at any given point of time, allowing for maximum parallelization. These results are equivalent to earlier work in SID-UP [Drechsler et al. 2014] and FElm [Czaplicki and Chong 2013] both of which are also optimal but proven here for the first time. We only consider configurations during propagation where the skip rule has already been fully applied, because skipping reactives does not constitute work we want to parallelize, and this simplifies the proof. SID-UP [Drechsler et al. 2014] shows an efficient way to not compute skips at all.

Lemma 4.14. *For a propagation $p = ptrace(D_0, D_n)$ and device $D_i \in p$ with processed reactives $P \in D_i$ and store $\mu \in D_i$. If the skip rule is not applicable to D_i, then reevaluating a reactive $r \notin ready(P, \mu)$ violates one of our correctness guarantees.*

Proof. By contradiction. Assume there exists a reactive $r \in dom(\mu)$ but $r \notin ready(P, \mu)$. It holds $r \notin P$, otherwise r would be reevaluated twice (Lemma 4.5). Inspecting the premises of the ready rule shows that there must be an input $r_i \in inputs(r, \mu)$ which is not yet processed $r_i \notin P$. We use a similar argument to Lemma 4.7 to show that either r_i or one of its predecessors must be ready. Due to complete propagation (Lemma 4.10) the reactive r_i will be reevaluated in the future because it has a ready predecessor. Thus, reevaluating r is not glitch-free (Lemma 4.6). □

4.8 Distributed Updates

Distributed reactives in $\mathcal{F}_R$ have a commutativity and idempotent merge function for each distributed reactive $r \in dom(M)$, and all devices use the same merge function for the same distributed reactive r. Thus, each distributed reactive corresponds to a CRDTs for which eventual consistency is well known [Jagadeesan and Riely 2018]. $\mathcal{F}_R$ combines of eventual consistency and complete propagation (Lemma 4.10) to provide eventual consistency for the interactive application. In general, state derived from a distributed reactive r could become inconsistent between replicas, because both replicase observe a different number and order of activations of r. The critical insight is that only the state of fold reactives depends on the number and order of activations. Thus, fold reactives derived from a distributed reactive must be distributed themselves to make the application eventually consistent. This property is captured by the following theorem.

Definition 4.15 (Consistent reactives). Given a reactive r and devices D_1 and D_2 with states $\mu_1 \in D_1$ and $\mu_2 \in D_2$, we say r is consistent if $r \notin dom(\mu_1) \cap dom(\mu_2)$ or $\mu_1(r).val = \mu_2(r).val$

Lemma 4.16 (Eventual consistency). *Given two devices D_1 and D_2 and reactive $r \in dom(M)$. If there are no other changes to r and the devices eventually exchange values, then r is consistent.*

Proof. Follows directly from commutativity and idempotence of $M(r)$ once both devices have received the remote value at least once with the receive rule. □

Theorem 4.17. *A fully connected component of reactives R in the dataflow graph is eventually consistent, if all sources $R_s \subset R$ and folds $R_f \subset R$ are eventually consistent.*

Proof. Once all I and F reach a consistent state, then because of complete propagation (Lemma 4.10), all derived reactives R become consistent. □

4.9 Restoration

Restoration consists of two parts: taking *proper* snapshots and restoring the application and its state. When a device starts for the first time the snapshot is empty, and when it is restored the snapshot was proper in a prior run. We will first show that the reduction rules on devices produce proper snapshots. We then show that, for recovery, using a proper snapshot restores the application.

A device is *proper*, if each value in the current snapshot matches the corresponding value in the store. The reduction relation $\rightarrow_D$ reduces configurations with proper devices to other proper devices, and propagations started at a proper device will result in a proper device.

Definition 4.18 (Proper device). A device $D = \Sigma \langle \mu \rangle\, t$ is proper if and only if all sources and folds in μ are included in Σ with $\forall r \in dom(\mu) : (inputs(r, \mu) = \emptyset \lor r \in inputs(r, \mu)) \rightarrow \mu(r).val = \Sigma(r)$.

Lemma 4.19 (Evaluation produces proper devices). *If $D = \Sigma \langle \mu \rangle\, t$ is proper, then $D' = \Sigma' \langle \mu' \rangle\, t'$ with $D \rightarrow^* D'$ is also proper.*

Proof. The source and fold rules ensure that created reactives are included in the snapshot, and at no other time is the store modified outside of an ongoing propagation. The commit rule updates the snapshot to include the values of all reactives which were active during propagation. □

For the remaining discussion, we individually look at two separate stages of executing the application: the first stage which only creates new reactives, which we call the creation prefix of the application, and the rest of the application which does read or modify the restored values from the dataflow graph. For example, the chat application in Figure 1 only consists of a creation part. Pure FRP languages such as FElm [Czaplicki and Chong 2013] only consist of such creation parts, and modifications or observations of the dataflow graph are only executed by the runtime after the program has terminated. We show that for the creation prefix, the restoration produces an exact subset of the state before a crash, and that the rest of the program has a trace which produces the same observable behavior as if the crash had not happened.

Definition 4.20 (Creation prefix). The creation prefix of a device, $ctrace(D_0, D_n)$, is the longest prefix $D_0 \rightarrow_p^* D_n$ of $trace(D_0)$ where the steps $\rightarrow$ was not produced by the fire or access rules.

Consider restoring just the creation prefix of an application. During restoration – because the application is deterministic – it reproduces the original creation trace. When the application has executed all creation reductions, its state is a proper subset of the state before the crash. All restored reactives have the same value they had before the crash. However, reactives created after the creation prefix have not yet been restored.

Lemma 4.21. *For any proper device $D \in trace(D_0)$ with $D = \Sigma\,\langle\mu\rangle\, t$ and $D_0 = \Sigma_0 L$ the re-execution of the creation trace of the application with the snapshot $ctrace(\Sigma L, D')$ ending with D' has a store $\mu' \in D'$ which agrees with the original store $\mu' \subseteq \mu$.*

Proof. Creation of reactives during restoration still happens in the same order, because there are no steps produced by the access rule in the creation prefix that, i.e., the execution of the process can never observe any differing values from the snapshot. In particular, $dom(\mu') \subseteq dom(\mu)$. It remains to show that each reactive $r \in dom(\mu')$ matches the value before the crash $\mu(r).val = \mu(r).val$. Due to proper devices, it holds that $\forall r \in dom(\Sigma) \cap dom(\mu) : \mu(r).val = \Sigma(r)$. For the source and fold rules, the lookup function returns $\Sigma(r)$ as a new value, which is equal to $\mu(r).val$. For the map and flatten rules, the new value is computed from the values of the dependencies, which is equal to $\mu(r).val$ because of complete propagation (Lemma 4.10). □

When continuing evaluation after the creation prefix, applications may trigger new actions or read the value of already restored reactives, which may be different compared to the original execution which was initialized with a different snapshot. Thus, reactives may be created in different orders, and firing of new actions may cause different reactions. We first show that the value of restored reactives is correct, independent of the order of restoration.

Lemma 4.22. *Assume a restoration of $ctrace(D'_0, D')$ as in Lemma 4.21. Creation of new reactives in the restored trace $D' \in trace(D'_0)$ will produce a store μ' compatible with the store μ of the crashed device D: $\forall r \in dom(\mu) \cap dom(\mu') : \mu(r) = \mu'(r)$.*

Proof. Lemma 4.21 does not depend on the order in which reactives are restored. Thus, the reactives are restored with the correct values, independent of the order of their restoration. □

Triggering new actions after restoring the creation prefix of the device D'_0 also causes compatible changes. That is, the same action propagated in the original device D and the restored device D' will change the same reactives consistently.

Lemma 4.23. *Assume a device D and the restored device D' as in Lemma 4.22, i.e., their stores $\mu \in D$ and $\mu' \in D'$ are compatible $\forall r \in dom(\mu) \cap dom(\mu') : \mu(r) = \mu'(r)$. Propagating the same action on both devices produces new devices $D \to_p^* D_p$ and $D' \to_p^* D'_p$ with stores $\mu_p \in D_p$ and $\mu'_p \in D'_p$ that are still compatible: $\forall r \in dom(\mu_p) \cap dom(\mu'_p) : \mu_p(r) = \mu'_p(r)$.*

Proof. Follows from complete propagation for the same store and action (Lemma 4.10). The updated restored store is still a subset of the updated store before the crash $\mu'_p \subseteq \mu_p$, as the same values have been updated in both stores. □

When propagating the same action on the original device D and the restored device D', the propagation may update reactives of the original device, which have not been created on the restored device. In particular, if a fold reactive is not yet restored, any action changing that fold on the original device may cause the applications to diverge. However, this problem occurs even when considering device evaluation without crashes. The problem occurs if the application allows an action to trigger before or after creating a derived fold, e.g., the application enables user input before the fold is created. Restoration reflects this behavior of folds.

In conclusion, snapshots contain enough state to restore the store of the application to the state it was before a crash. In addition, $\mathcal{F}_R$ allows actions to change the state for a partially restored application, thus applications immediately react to actions without compromising correctness.

Theorem 4.24 (Restoration). *The creation prefix of the execution of an application is restored exactly as before a crash, and after the creation prefix any action causes the same changes as before the crash.*

Proof. Follows from Lemma 4.21, Lemma 4.22, and Lemma 4.23. □

The actions after a crash cause a divergence of the state before and after restoration. We assume this behavior of restoration to be intuitive to programmers, as it is very similar to how the language behaves without fault tolerance when actions are triggered during the initialization of the application. Applications which require more strict correctness guarantees must initialize their dataflow graph as part of the creation prefix, which guarantees exact recovery. When an application is recovered, the eventually consistent distribution mechanism will ensure that the application is again brought up-to-date.

5 EVALUATION

Having formally proven the consistency guarantees provided by $\mathcal{F}_R$, we now provide evidence that its programming model supports suitable abstractions to implement an existing interactive application. We implemented $\mathcal{F}_R$ as an open source (APL-2) Scala library within the REScala [Salvaneschi et al. 2014] ecosystem[1]. This library is then used to implement the TodoMVC[2] case study and extend it with peer-to-peer data synchronization. We first discuss the benefits on the application code when fault tolerance is part of the programming model. We then compare the performance of our TodoMVC implementation with two other existing implementations that also add data synchronization between multiple client instances.

5.1 Implementation

We implemented $\mathcal{F}_R$ within REScala, a reactive programming library that provides syntax and semantics of local reactives. The implementation integrates snapshot and recovery mechanisms proposed by Mogk et al. [2018], network communication packages from Weisenburger et al. [2018], and a custom CRDT library based on Shapiro et al. [2011a]. The programming API of the implementation and formalization have minor differences. Instead using of λ-calculus, user-defined operators are written in a functional subset of Scala. The implementation of $\mathcal{F}_R$ benefits from Scala's type checker, i.e., operator types must match their inputs. The implementation also provides operators for common usage patterns, e.g., `count` for a `Fold` that counts the number of activations.

The formalization does not specify the generation of fresh names, serialization, network communication and connectivity. These depend on the concrete implementation of the application and are the responsibility of the developer. However, we do provide the solutions we developed in our case study for all these aspects as readily reusable components. REScala automatically generates unique names for each reactive, which is used for the locations in $\mathcal{F}_R$. The developer only must ensure that all distributed reactives that should be connected with each other have the same name, usually by specifying these names manually. Serialization is delegated to existing libraries, which handle serialization of simple data structures automatically. Developers only need to manually implement serializers for complex custom data structures, in particular for data structures with nested reactives. The desired type of network communication varies greatly between applications, from a web application using JSON messages over Websockets to a server exchanging binary data on a TCP socket. For maximum compatibility, $\mathcal{F}_R$ provides a simple interface to integrate with any bidirectional communication channel.

[1] http://www.rescala-lang.com/
[2] http://todomvc.com/

5.2 Case Study

TodoMVC is a to-do list application that is widely used to compare different languages and frameworks for programming interactive applications. The TodoMVC application shows a list of tasks, each representing one to-do item. Tasks can be added, changed, completed and removed. At the time of writing, the official website of TodoMVC presents 64 implementations in different languages and frameworks, and many more unofficial implementations (such as ours) exist. Synchronization of data across multiple clients is not part of the official feature set of TodoMVC, thus not all implementations support synchronization. However, we were able to find two implementations that do support synchronization with slightly different guarantees and use them as a comparison to $\mathcal{F}_R$. One uses Twilio[3] to synchronize state with a centralized server, providing strong consistency, but prohibiting offline usage. The other uses Flask[4] to synchronize state with a central server, but does not make any guarantees, i.e., changes may arbitrarily get lost. We discuss our implementation of TodoMVC and the other two before comparing them.

The $\mathcal{F}_R$ implementation. To synchronize the full state of the application between devices, we implement the list of tasks and each individual task as distributed reactives. Each task has a unique identifier and uses a last-writer-wins CRDT [Shapiro et al. 2011a]. Last-writer-wins is appropriate since a task represents a single data item. Each task has its own "done" button and "edit" input box, which are the sources from which the task is derived. Various parts of the UI are derived from each task reactive. The list of tasks is a single distributed fold with a fixed name. The list is stored as a replicated growable array (RGA [Shapiro et al. 2011a]), because tasks are added and removed at any position. As the list contains nested reactives (the tasks), the RGA requires a custom user-defined serializer. This serializer stores the ID and the initial value of each task. During deserialization, the application uses the ID to look up the corresponding task reactive. If no task exists for a given ID, a new one is created with the given ID and initial value, using the same function that is used when the user presses a button to create a new task.

Twilio and Flask Implementations. Both implementations support recovery based on a central server – there is no support for merging diverged states. Thus, the application is unavailable when the device is offline. The two approaches differ their approach to state synchronization.

Flask replaces the local storage used by TodoMVC with a custom implementation that has a compatible interface. The result is that new or modified tasks are individually sent to a central server, which orders based on arrival time. The application only requests state from the server during startup. That is, changes to the to-do list on one device become visible on other devices only when the application is restarted there, resulting in no consistency guarantees for the user.

Twilio has its own commercial implementation of a replicated list. The list has a primary replica on a central server, which ensures consistency of the list, but it requires local replicas to request confirmation for each change. Thus, Twilio ensures consistency at the cost of not supporting offline usage. Also, once the Twilio implementation displays more than 50 tasks at once, multiple devices become inconsistent when removing tasks, most likely due to an implementation bug in the application's UI.

5.3 Effect on Application Code

Our implementation of TodoMVC consists of 252 lines of code. Most of the code expresses the application logic in the programming model of $\mathcal{F}_R$ and is not specific to distribution or fault-tolerance. Out of all lines, only 5 lines of code accommodate the distribution of individual tasks by

[3] https://www.twilio.com/blog/2017/09/building-a-todomvc-with-twilio-sync.html
[4] https://simplectic.com/blog/2014/flask-todomvc-backbone-sync/

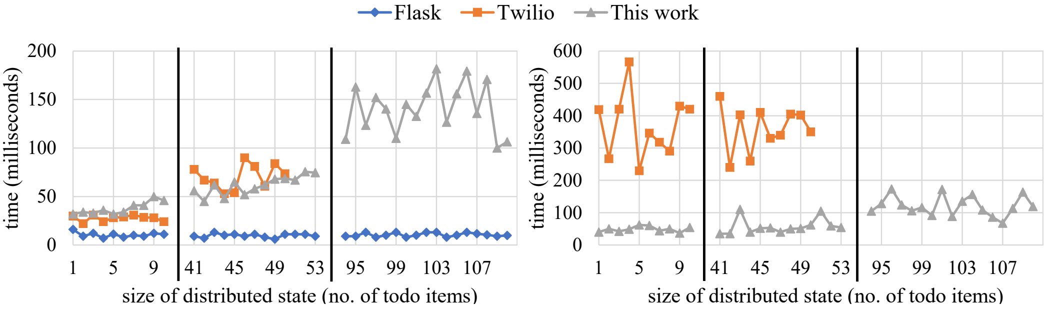

Fig. 10. Time spent on computation (left) and network (right) when adding a new task.

using CRDTs instead of plain data. Another 5 lines of code are required for the task list, including the custom serializer implementation mapping each task to its ID and initial value, the ID lookup, and the creation of new tasks during deserialization. The rest of the application then automatically reacts to changes from remote devices.

The Flask implementation has 268 lines of code. The implementation is based on one of the official versions of TodoMVC, and only modifies 4 lines of code to exchange the local storage back end used by TodoMVC with the one used by Flask. However, the storage interface was not designed to support fault tolerance, which makes it impossible for the framework to provide any guarantees in its alternative implementation. For example, the local storage API is not designed for values to be changed outside of the application, thus the API does not provide any means to notify the application of remote changes. Even if notification were available, handling them correctly would require restructuring the rest of the application.

The Twilio implementation has 818 lines of code. Out of these, 14 deal with authenticating with their central server, 32 implement routing for callback listeners when tasks are modified, and 70 implement the various interactions with the custom API of their distributed data structure. This boilerplate code is specific to this application, thus the functionality it supports cannot be easily moved to a library implementation.

To recap, our case study indicates that implementing fault-tolerant interactive distributed applications with eventual consistency guarantees in $\mathcal{F}_R$ is not only feasible, but also does not increase the complexity of the application logic in order to make it fault-tolerant.

5.4 Performance

Experimental setup. We use the Chromium developer tools[5] to collect performance data and use reported script execution time as a measurement of computational overhead, and the idle time between events as a measure for network communication latency. All network connections happen in a gigabit LAN with less than 1 millisecond latency. The exception is the Twilio implementation, which uses its own external commercial service. We measured 1 to 2 ms latency to the Twilio cloud front servers, but we have no insight on how much network latency their API requests incur internally.

Results. We study the performance of adding a single new task to task lists of different sizes. For $\mathcal{F}_R$, we use a peer-to-peer connection between two devices running the same application. Even though $\mathcal{F}_R$ does not have to wait for remote confirmation to apply updates, we still measure the time until the local device receives a confirmation that the update has been applied remotely.

[5] Chromium 73.0.3683.75, Ubuntu 18.10 (64-bit), Intel(R) Core(TM) i5-5300U

Figure 10 shows the results of the experiment. The x-axis shows the size of the task list at which the operation happened, and the y-axis the time taken to add a single new task. The left graph shows the time spent on local computations. The right graph shows the round-trip time of sending tasks to the remote device, including processing time of the remote device.

The Flask implementation has the least computational overhead and the overhead is independent of the size of the list. Flask provides no fault-tolerance and does not confirm that remote changes are applied, thus there is no network overhead to measure. The Twilio-based implementation is on the other extreme of the spectrum. When making changes, users must wait for both computation and communication to finish, because clients eventually receive a confirmation. In our case, this approach results in an average of 412 ms until any interaction produces a result. Note that the exact numbers heavily depend on a remote system outside of our control.

In contrast to the Twilio-based implementation, $\mathcal{F}_R$ can display changes to user inputs without waiting for confirmation. Therefore, changes become visible after 109ms on average, which consists only of computational overhead. The computational overhead of $\mathcal{F}_R$ grows with the size of the task list because the list is stored in a replicated growable array (RGA), which has the most overhead of all CRDTs in our implementation. Concretely, our RGA implementation has an overhead of 357 bytes for added tasks, and 224 bytes for deleted tasks. However, the overhead is independent of the content of the tasks, i.e., of the actual size of application level objects, as those are synchronized separately. In terms of network communication, waiting for confirmation from the remote device requires a similar amount of time as a local change. This is expected, as the remote device mirrors the behavior of the local device before sending the confirmation.

Summary. Our results show that while consistent approaches do have an impact on performance, the eventual consistent approach of $\mathcal{F}_R$ has competitive computational cost compared to the strong consistency of Twilio, which gives up availability. This is a very good result given that dealing with unreliable communication and crashes has a significant performance impact in the implementation of $\mathcal{F}_R$. Moreover, performance in $\mathcal{F}_R$ does not depend on network conditions.

The performance of $\mathcal{F}_R$ is affected noticeably by the size of remotely shared data, compared to an approach that is independent of data sizes such as Flask. However, this is to a significant extent due to $\mathcal{F}_R$ implementation still being in its infancy, with clear directions for improvement. In future work, we plan to improve the performance of $\mathcal{F}_R$ in two ways. First, currently serialization happens independently for snapshots and communication, which enables decoupling those features in the implementation, but causes duplicate computation. We can address the problem by serializing each value only once and using it both for the snapshot and communication. This approach would result in saving 23ms in the best case for lists of size 100. Second, further improvements are possible by synchronizing modifications to list-style CRDTs incrementally (e.g., using the approach by Almeida et al. [2018]).

6 RELATED WORK

$\mathcal{F}_R$ is related to work in the areas of cluster-based systems, interactive applications, formal programming models for concurrent and distributed computations, and abstractions for unreliable networks. We consider strengths and weaknesses of work in these areas but find all of them lacking some desirable properties for distributed interactive applications with decentralized state.

6.1 Cluster-Based Systems

Due to the large number of devices involved in clusters, failures of at least one device become frequent. However, there are ample spare resources to replace failed devices. In MapReduce [Dean and Ghemawat 2008; Lämmel 2008] the severely restricted programming model facilitates rescheduling of tasks. Consistency is ensured by a central coordinator. Dryad [Isard and Yu 2009], PigLatin [Olston et al. 2008], and FlumeJava [Chambers et al. 2010] are similar to MapReduce - the difference is in their support for more flexible operations than just map and reduce. Spark [Zaharia et al. 2012], Flink [Alexandrov et al. 2014; Carbone et al. 2015], and MBrace [Dzik et al. 2013] express computations as embedded domain specific functional dataflow languages. Viering et al. [2018] develop a typing discipline to ensure correct execution even in the presence of faults. None of these systems is applicable without central coordination and replacements for failed devices, which makes sense in their environment, but not for interactive applications. Bernstein et al. [2017] introduce an eventually consistent mechanism to synchronize state between multiple data centers, however, the solution does not extend to client devices. GSP [Burckhardt et al. 2015] provides a semantic foundation for eventual consistency using a central server connected to clients with unreliable connections; GOS [Gotsman and Burckhardt 2017] generalizes GSP and provides a formal foundation for proving equivalence of multiple implementations. GSP operates at the edge, it requires a central system for ordered message delivery, but enables eventual consistency for client devices. Thus, GSP can be integrated with $\mathcal{F}_R$ for distributed reactive using GSP instead of CRDTs.

6.2 Reactive and Interactive Systems

Functional reactive programming (FRP) originally considers systems with continuous time [Elliott and Hudak 1997] but has inspired many languages that like $\mathcal{F}_R$ handle discrete changes [Cooper and Krishnamurthi 2006; Czaplicki and Chong 2013; Kamina and Aotani 2018]. Florence et al. [2019] provide a calculus for Esterel – a synchronous programming language conceptually similar to discrete FRP, but with very different syntax. DREAM [Margara and Salvaneschi 2014, 2018] analyze consistency levels of different propagation algorithms based on a formal model of FRP like propagation of events in a distributed system. Jeffrey [2013a,b, 2014] shows that a type system for FRP has a Curry-Howard [Howard 1980] correspondence to linear time logic. ScalaLoci [Weisenburger et al. 2018] combines distributed FRP with static typing for remote communication. None of these systems considers faults. REScala [Salvaneschi et al. 2014] extends FRP with thread-safety, distribution, and fault tolerance [Drechsler et al. 2018, 2014; Mogk et al. 2018]. However, the interaction between distribution, propagation, and faults has not been considered, especially not based on a rigorous formal language model with formal guarantees about the properties of the overall system.

Actors [Akka 2019b; Armstrong 2010; Bernstein et al. 2014; Karmani and Agha 2011; Miller et al. 2005; Van Cutsem et al. 2014] are well-known abstractions for loosely coupled distributed systems. Actor systems always consider faults but are too unstructured to provide fully automatic solutions. Akka [2019b] and Erlang [Armstrong 2010] have supervision hierarchies which allow user-defined handlers to redeploy actors on crashes. Orleans [Bernstein et al. 2014] and extensions to Akka [2019a] provide automatic restoration of individual actor state in cluster environments but provide no overall system consistency. Reactive Caching [Burckhardt and Coppieters 2018] adds a layer for push-based state for client devices on top of Orleans. AmbientTalk [Van Cutsem et al. 2014] extends actors with service discovery for mobile ad-hoc networks. Direst [Myter et al. 2016] builds on top of AmbientTalk and adds reactive abstractions with synchronization similar to $\mathcal{F}_R$ but requires a central replica. These systems either require a cluster like environment (with central coordination) or delegate the responsibility for ensuring consistency in case of faults to the application.

6.3 Formal Models for Concurrent and Distributed Computations

There are a wide variety of calculi [Cardelli and Gordon 1999; Fournet and Gonthier 1996; Hoare 1978; Milner 1982; Milner et al. 1992] covering traditional distributed and concurrent systems. These calculi focus on systems with multiple processor cores, thus they have no inherent concept of unreliable communication over the Internet. Caires and Pérez [2017] adds linear types to the π-calculus, to model protocols including unreliable message delivery. Dal Lago et al. [2019] investigate theoretical foundations for typed runtime errors in the π-calculus. Cloud calculus [Jarraya et al. 2012] focuses on upholding low-level security policies when migrating processes between virtual machines. The above systems address reliability (if at all) for individual internal messages, not providing application-level guarantees. More recently, Atkey [2017] adds external communication to classical processes, with a focus on the external high-level behavior, bringing the abstraction level closer to our goals for $\mathcal{F}_R$. v. Gleissenthall et al. [2019] extracts typical communication structures from programs with only minimal annotations, and generates a synchronous model for those communications for which proofs apply to the original asynchronous program. This extraction fits the synchronous model of $\mathcal{F}_R$, so future could apply their analysis.

CPL [Bračevac et al. 2016] is a calculus to create and combine multiple services in a type safe manner. Function passing [Haller et al. 2018; Miller et al. 2016] defines derived values of immutable remote data in a type safe way. Concurrent Objects [Filipović et al. 2010] provides objects with serializability and linearizability by guaranteeing that concurrent implementations behave observational equivalent to sequential implementations. Attiya et al. [2013] provides transactional behavior on the abstraction level of typical programming APIs. None of these languages consider faults beyond allowing application defined fault handling, thus providing no automatic guarantees.

6.4 Abstractions for Unreliable Networks

Hellerstein and Alvaro [2019] argue that monotonic data structures work well in distributed systems, because they represent the class of applications, which can be available and consistent [Hellerstein 2010]. Eventual consistent data types such as CRDTs [Shapiro et al. 2011a] or CloudTypes [Burckhardt et al. 2012] are important building blocks providing well-understood tradeoffs between consistency and responsiveness. Lasp [Meiklejohn and Van Roy 2015a,b] tackles the problem of deriving one CRDT from another, preventing multiple derivations to produce duplicates. LVars [Kuper and Newton 2013] uses monotonicity for deterministic concurrency and has been combined with CRDTs to provide determinism and distribution [Kuper and Newton 2014]. Bloom [Conway et al. 2012] supports declarative composition of monotonic operations to write monotonic programs. All of the above restrict the application to use only monotonic operations, which is not required for local consistency. $\mathcal{F}_R$ only uses monotonic operations for distribution but allows arbitrary operations locally.

Myter et al. [2018] enable switching between strong consistent and eventual consistent communication in a single language, if the application is tolerant to unavailability in case of connection problems. Meiklejohn [2017] argues for the need of languages to support both forms of synchronization, based on earlier arguments by Landin [1966]. Zhang et al. [2014] argue that many traditional applications become distributed, thus they enable a flexible model, where distribution details are specified by different libraries for each group of objects. Goldstein et al. [2018] argues for using database techniques to provide durability for general purpose applications, by logging all incoming actions for the application and replaying the log in case of failures. Combinations of these proposals with the $\mathcal{F}_R$ model seem worth considering, given that the dataflow graph and the semantics of propagation of $\mathcal{F}_R$ provide much of the insights and programming disciplines these proposals require.

7 CONCLUSION

Decentralized architecture will only become more relevant in the future, as they improve resource usage, latency, and privacy in distributed applications. Their failure modes, such as device crashes and disrupted communication, are unavoidable, meaning distributed interactive applications must address them. Current solutions require reasoning about low level properties of message propagation, and ensuring consistency requires too much commitment from developers.

In this paper, we presented $\mathcal{F}_R$, a programming model for distributed interactive applications that combines the best of two worlds. On the one hand, $\mathcal{F}_R$ is practical in that it provides an elegant, high-level programming model with automatic local and distributed change propagation and automatic recovery from crashes and communication disruptions while guaranteeing strong eventual consistency. Its implementation is readily available in the form of a Scala library, and has promising performance. On the other hand, $\mathcal{F}_R$ has formally proven correct semantics, and therefore also enables formal reasoning on the theoretical side about the behavior of applications, including propagation, consistency, and recovery from crashes or disconnects.

ACKNOWLEDGMENTS

We thank the reviewers of this paper, their valuable feedback was very helpful to improve the paper. We thank Alexander Kopp for contributing the first version of the operational semantics and properties of propagation. This work is supported by the German Research Foundation (DFG, No. 415626024, 383964710 and 322196540, SFB 1053 and 1119), by the European Research Council (ERC, Advanced Grant No. 321217 and 862535), and by the LOEWE initiative in Hessen, Germany (HMWK, NICER, emergenCITY, Software-Factory 4.0).

REFERENCES

Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

Akka. 2019a. Distributed Data Clusters. https://doc.akka.io/docs/akka/current/distributed-data.html.

Akka. 2019b. Documentation. http://akka.io/docs.

Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere platform for big data analytics. *The VLDB Journal* 23, 6 (01 Dec 2014), 939–964. https://doi.org/10.1007/s00778-014-0357-y

Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta state replicated data types. *J. Parallel and Distrib. Comput.* 111 (2018), 162 – 173. https://doi.org/10.1016/j.jpdc.2017.08.003

Joe Armstrong. 2010. Erlang. *Commun. ACM* (Sept. 2010). https://doi.org/10.1145/1810891.1810910

Robert Atkey. 2017. Observed Communication Semantics for Classical Processes. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer, Berlin, Heidelberg, 56–82.

Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. 2013. A Programming Language Perspective on Transactional Memory Consistency. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC '13).* ACM, New York, NY, USA, 309–318. https://doi.org/10.1145/2484239.2484267

P. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Technical Report. (MSR-TR-2014-41, 24). http://aka.ms/Ykyqft

Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M. Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. 2017. Geo-distribution of Actor-based Services. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 107 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133931

Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.* 19, 2 (Nov. 1992). https://doi.org/10.1016/0167-6423(92)90005-V

Oliver Bračevac, Sebastian Erdweg, Guido Salvaneschi, and Mira Mezini. 2016. CPL: A Core Language for Cloud Computing. In *Proceedings of the 15th International Conference on Modularity (MODULARITY 2016).* ACM, New York, NY, USA, 94–105. https://doi.org/10.1145/2889443.2889452

Sebastian Burckhardt and Tim Coppieters. 2018. Reactive Caching for Composed Services: Polling at the Speed of Push. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 152 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276522

Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. 2012. Cloud Types for Eventual Consistency. In *ECOOP 2012 – Object-Oriented Programming*, James Noble (Ed.). Springer, Berlin, Heidelberg, 283–307. https://doi.org/10.1007/978-3-642-31057-7_14

Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*

(Leibniz International Proceedings in Informatics (LIPIcs)), John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 568–590. https://doi.org/10.4230/LIPIcs.ECOOP.2015.568

Luís Caires and Jorge A. Pérez. 2017. Linearity, Control Effects, and Behavioral Types. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer, Berlin, Heidelberg, 229–259. https://doi.org/10.1007/978-3-662-54434-1_9

Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. 2015. Lightweight Asynchronous Snapshots for Distributed Dataflows. *CoRR* abs/1506.08603 (2015). arXiv:1506.08603 http://arxiv.org/abs/1506.08603

Luca Cardelli and Andrew D. Gordon. 1999. Types for Mobile Ambients. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 79–92. https://doi.org/10.1145/292540.292550

Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: Easy, Efficient Data-parallel Pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 363–375. https://doi.org/10.1145/1806596.1806638

Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and Lattices for Distributed Programming. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*. ACM, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/2391229.2391230

Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-value Language. In *Proceedings of the 15th European Conference on Programming Languages and Systems (ESOP'06)*. Springer-Verlag, Berlin, Heidelberg, 294–308. https://doi.org/10.1007/11693024_20

Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY, USA, 411–422. https://doi.org/10.1145/2491956.2462161

Ugo Dal Lago, Marc de Visme, Damiano Mazza, and Akira Yoshimizu. 2019. Intersection Types and Runtime Errors in the Pi-calculus. *Proc. ACM Program. Lang.* 3, POPL, Article 7 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290320

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. 2018. Thread-safe Reactive Programming. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 107 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276477

Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 361–376. https://doi.org/10.1145/2660193.2660240

Jan Dzik, Nick Palladinos, Konstantinos Rontogiannis, Eirik Tsarpalis, and Nikolaos Vathis. 2013. MBrace: Cloud Computing with Monads. *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems* (2013). https://doi.org/10.1145/2525528.2525531

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 263–273. https://doi.org/10.1145/258948.258973

V. Enes, P. S. Almeida, C. Baquero, and J. Leitão. 2019. Efficient Synchronization of State-Based CRDTs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 148–159. https://doi.org/10.1109/ICDE.2019.00022

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.* 103, 2 (Sept. 1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for concurrent objects. *Theoretical Computer Science* 411, 51 (2010), 4379 – 4398. https://doi.org/10.1016/j.tcs.2010.09.021 European Symposium on Programming 2009.

Spencer P. Florence, Shu-Hung You, Jesse A. Tov, and Robert Bruce Findler. 2019. A Calculus for Esterel: If Can, Can. If No Can, No Can. *Proc. ACM Program. Lang.* 3, POPL, Article 61 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290374

Cédric Fournet and Georges Gonthier. 1996. The Reflexive CHAM and the Join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 372–385. https://doi.org/10.1145/237721.237805

Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *ACM SIGACT News* 33, 2 (June 2002). https://doi.org/10.1145/564585.564601

Jonathan Goldstein, Ahmed Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Umar Farooq Minhas, Ryan Newton, Rahee Ghosh Peshawaria, Tal Zaccai, and Irene Zhang. 2018. *A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications.* Technical Report. https://www.microsoft.com/en-us/research/publication/a-m-b-r-o-s-i-a-providing-performant-virtual-resiliency-for-distributed-applications/

Alexey Gotsman and Sebastian Burckhardt. 2017. Consistency Models with Global Operation Sequencing and their Composition. In *31st International Symposium on Distributed Computing (DISC 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Andréa W. Richa (Ed.), Vol. 91. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 23:1–23:16. https://doi.org/10.4230/LIPIcs.DISC.2017.23

Philipp Haller, Heather Miller, and Normen Mueller. 2018. A programming model and foundation for lineage-based distributed computation. *Journal of Functional Programming* 28 (2018), e7. https://doi.org/10.1017/S0956796818000035

Joseph M. Hellerstein. 2010. The Declarative Imperative: Experiences and Conjectures in Distributed Logic. *SIGMOD Rec.* 39, 1 (Sept. 2010), 5–19. https://doi.org/10.1145/1860702.1860704

Joseph M. Hellerstein and Peter Alvaro. 2019. Keeping CALM: When Distributed Consistency is Easy. *CoRR* abs/1901.01930 (2019). arXiv:1901.01930 http://arxiv.org/abs/1901.01930

C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677. https://doi.org/10.1145/359576.359585

William A Howard. 1980. The formulae-as-types notion of construction.

Michael Isard and Yuan Yu. 2009. Distributed Data-parallel Computing Using a High-level Programming Language. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 987–994. https://doi.org/10.1145/1559845.1559962

Radha Jagadeesan and James Riely. 2018. Eventual Consistency for CRDTs. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 968–995. https://doi.org/10.1007/978-3-319-89884-1_34

Y. Jarraya, A. Eghtesadi, M. Debbabi, Y. Zhang, and M. Pourzandi. 2012. Cloud calculus: Security verification in elastic cloud computing platform. In *2012 International Conference on Collaboration Technologies and Systems (CTS)*. 447–454. https://doi.org/10.1109/CTS.2012.6261089

Alan Jeffrey. 2013a. Causality for Free!: Parametricity Implies Causality for Functional Reactive Programs. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification (PLPV '13)*. ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/2428116.2428127

Alan Jeffrey. 2013b. Functional Reactive Programming with Liveness Guarantees. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/2500365.2500584

Alan Jeffrey. 2014. Functional Reactive Types. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (CSL-LICS '14)*. ACM, New York, NY, USA, Article 54, 9 pages. https://doi.org/10.1145/2603088.2603106

Tetsuo Kamina and Tomoyuki Aotani. 2018. Harmonizing Signals and Events with a Lightweight Extension to Java. *Programming Journal* 2, 3 (2018), 5. https://doi.org/10.22152/programming-journal.org/2018/2/5

Rajesh K. Karmani and Gul Agha. 2011. Actors. In *Encyclopedia of Parallel Computing*. https://doi.org/10.1007/978-0-387-09766-4_125

Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. *Local-first software: You own your data, in spite of the cloud.* Web Publication. Ink & Switch. https://www.inkandswitch.com/local-first.html

Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece.*

Lindsey Kuper and Ryan R. Newton. 2013. LVars: Lattice-based Data Structures for Deterministic Parallelism. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC '13)*. ACM, New York, NY, USA, 71–84. https://doi.org/10.1145/2502323.2502326

Lindsey Kuper and Ryan R. Newton. 2014. Joining Forces Toward a Unified Account of LVars and Convergent Replicated Data Types. *WoDet* (2014).

Ralf Lämmel. 2008. Google's MapReduce programming model — Revisited. *Science of Computer Programming* 70, 1 (2008), 1 – 30. https://doi.org/10.1016/j.scico.2007.07.001

P. J. Landin. 1966. The Next 700 Programming Languages. *Commun. ACM* 9, 3 (March 1966), 157–166. https://doi.org/10.1145/365230.365257

Alessandro Margara and Guido Salvaneschi. 2014. We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. ACM, New York, NY, USA, 142–153. https://doi.org/10.1145/2611286.2611290

Alexandro Margara and Guido Salvaneschi. 2018. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Transactions on Software Engineering* 44, 7 (July 2018), 689–711. https://doi.org/10.1109/TSE.2018.2833109

Christopher Meiklejohn and Peter Van Roy. 2015a. The Implementation and Use of a Generic Dataflow Behaviour in Erlang. In *Proceedings of the 14th ACM SIGPLAN Workshop on Erlang (Erlang 2015)*. ACM, New York, NY, USA, 39–45. https://doi.org/10.1145/2804295.2804300

Christopher Meiklejohn and Peter Van Roy. 2015b. Lasp: A Language for Distributed, Coordination-free Programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP '15)*. ACM, New York, NY, USA, 184–195. https://doi.org/10.1145/2790449.2790525

Christopher S. Meiklejohn. 2017. On the Design of Distributed Programming Models. In *Proceedings of the Programming Models and Languages for Distributed Computing (PMLDC '17)*. ACM, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/3166089.3166093

Heather Miller, Philipp Haller, Normen Müller, and Jocelyn Boullier. 2016. Function Passing: A Model for Typed, Distributed Functional Programming. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. https://doi.org/10.1145/2986012.2986014

Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. 2005. Concurrency Among Strangers. In *Proc. Int. Symp. on Trustworthy Global Computing*. https://doi.org/10.1007/11580850_12

R. Milner. 1982. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg.

Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *Inf. Comput.* 100, 1 (Sept. 1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. 2018. Fault-tolerant Distributed Reactive Programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:26. https://doi.org/10.4230/LIPIcs.ECOOP.2018.1

Florian Myter, Tim Coppieters, Christophe Scholliers, and Wolfgang De Meuter. 2016. I Now Pronounce You Reactive and Consistent: Handling Distributed and Replicated State in Reactive Programming. In *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2016)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/3001929.3001930

Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2018. A CAPable Distributed Programming Model. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2018)*. ACM, New York, NY, USA, 88–98. https://doi.org/10.1145/3276954.3276957

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 1099–1110. https://doi.org/10.1145/1376616.1376726

Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2577080.2577083

Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. 2017. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Trans. Softw. Eng.* 43, 12 (Dec. 2017), 1125–1143. https://doi.org/10.1109/TSE.2017.2655524

Marc Shapiro, Annette Bieniusa, Nuno M. Preguiça, Valter Balegas, and Christopher Meiklejohn. 2018. Just-Right Consistency: reconciling availability and safety. *CoRR* abs/1801.06340 (2018). arXiv:1801.06340 http://arxiv.org/abs/1801.06340

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://hal.inria.fr/inria-00555588

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer, Berlin, Heidelberg, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend Synchrony: Synchronous Verification of Asynchronous Distributed Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 59 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290372

Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. 2014. AmbientTalk: Programming Responsive Mobile Peer-to-peer Applications with Actors. *Computer Languages, Systems & Structures* 40, 3-4 (Oct. 2014). https://doi.org/10.1016/j.cl.2014.05.002

Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu, and Lukasz Ziarek. 2018. A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 799–826. https://doi.org/10.1007/978-3-319-89884-1_28

Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. 2019. Replication-aware Linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 980–993. https://doi.org/10.1145/3314221.3314617

Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 129 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276499

W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang. 2018. A Survey on the Edge Computing for the Internet of Things. *IEEE Access* 6 (2018), 6900–6919. https://doi.org/10.1109/ACCESS.2017.2778504

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, and Henry M. Levy. 2014. Customizable and Extensible Deployment for Mobile/Cloud Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 97–112. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhang